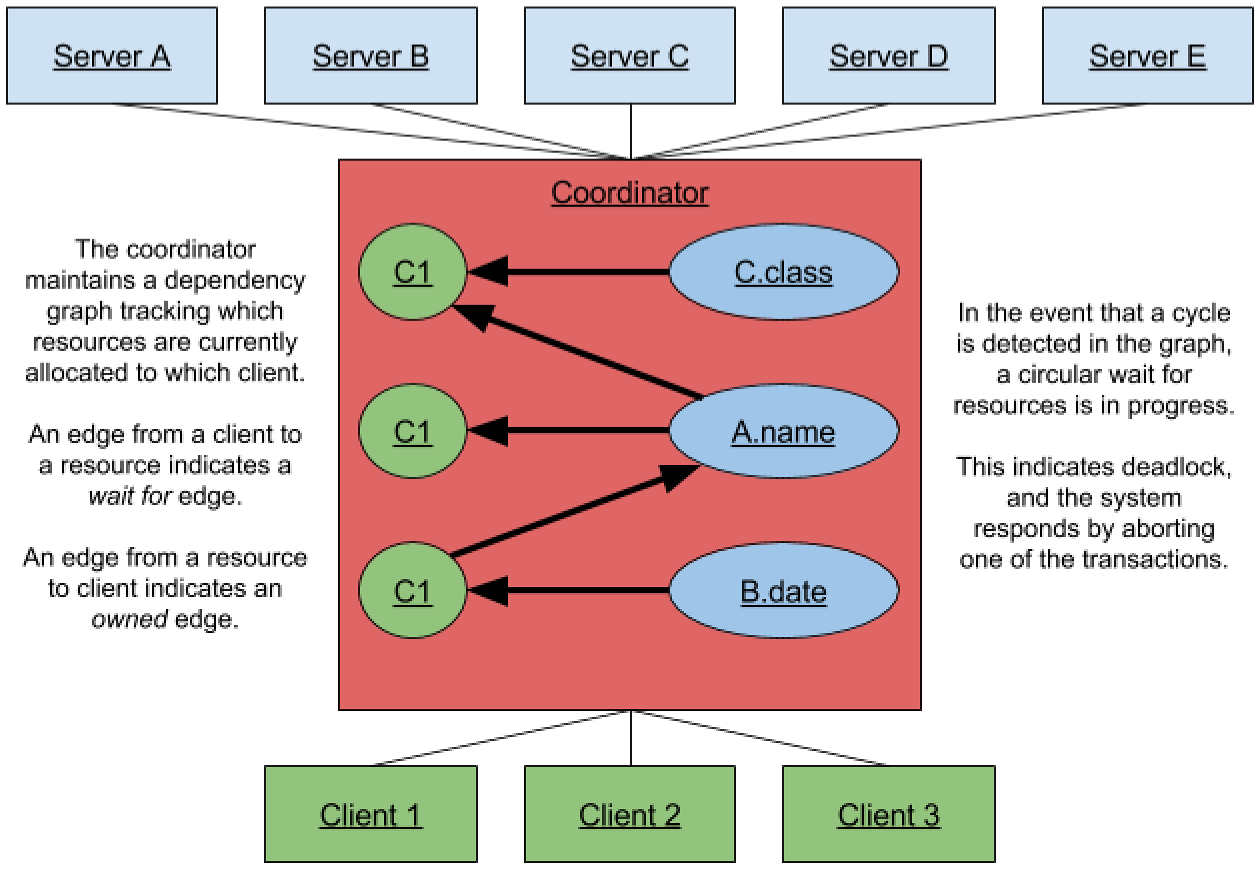
Parker Drake (ptdrake2)
Arvind Kouta (kouta2)

CS425 MP3 Design Document

Introduction

In this MP, we implemented a distributed transaction system. It supports transaction operations such as *get, set, commit,* and *abort*, and is capable of detecting and breaking deadlock. The system is consistent and transactions are isolated. This implementation makes use of read/write locks, remote function invocation, and a coordinator node to achieve these goals.

System Model and Control Flow


Server A
Server B
Server C
Server D
Server E
Coordinator
C1
C.class
C1
A.name
C1
B.date
The coordinator maintains a dependency graph tracking which resources are currently allocated to which client.
An edge from a client to a resource indicates a *wait for* edge.
An edge from a resource to client indicates an *owned* edge.
In the event that a cycle is detected in the graph, a circular wait for resources is in progress.
This indicates deadlock, and the system responds by aborting one of the transactions.
Client 1
Client 2
Client 3

Design

To modularize the code and improve debugging, the code was split into several classes.

1. Parser - This class handles all of the parsing of input from the client, and handles the responses from the coordinator and the servers.
2. Unlocker - This class handles deadlock breaking and resource allocation. It includes a graph implementation that can detect cycles in addition to a store of semaphores, one for each key/value pair in the system.
3. Main - On the servers, this class holds the actual key value stores operated on by the client. For the client, class simply loops accepting input. Main also holds the implementation for numerous helper RPC functions.

Deadlock Detection

The deadlock detection system uses a single coordinator node that synchronizes access to key value pairs. When a transaction attempts a *set* or a *get* on a key, the coordinator adds the appropriate edge to the dependency graph. Edges can be *read* or *write*, and can be *wait-for* or *owned* edges.

If the client wants *read* access to the variable, they are allowed access if no other transactions are writing. If a client wants *write* access to a variable, they are granted access only if no other transactions are reading or writing. In the event that a transaction cannot be immediately granted access to a key, a *wait-for* edge is installed, and the transaction will block. If any cycles are detected in the graph, it indicates circular wait, which must be broken to prevent deadlock. This is done by aborting a transaction.